RESEARCH

Categories for (Big) Data Models and Optimization

Laurent Thiry[??], Heng Zhao[1] and Michel Hassenforder[1]
[1]IRIMAS, Université de Haute Alsace
12, Rue des frères Lumière - 68093 Mulhouse (France)
{laurent.thiry, heng.zhao, michel.hassenforder}@uha.fr

[??]Correspondence:
laurent.thiry@uha.fr
IRIMAS, Université de Haute Alsace, 12, Rue des frères Lumière, 68093, Mulhouse, France
Full list of author information is available at the end of the article

Abstract

This paper proposes a theoretical foundation for Big Data. More precisely, it explains how "functors", a concept coming from Category Theory, can serve to model the various data structures commonly used to represent (large) data sets, and how "natural transformations" can formalize relations between these structures. Algorithms, such as querying a precise information, mainly depend on the data structure considered, and thus natural transformations can serve to optimize these algorithms and get a result in a shorter time. The paper details four functors modeling tabular data, graph structures (e.g. triple stores), cached and split data. Next, the paper explains how, by considering a functional programming language, the concepts can be implemented without effort to propose new tools (e.g. efficient information servers and query languages). And, as a complement to the mathematical models proposed, the paper also presents a optimized data server and a specific query language (based on "unification" to facilitates the search of information). Finally, the paper gives a comparison study and shows that this tool is more efficient than most of the standards available in the market: the functional server appears to be 10+ times faster than relational or document oriented databases (Mysql and MongoDB), and 100+ times faster than a graph database (Neo4j).

Keywords: Data models; Category Theory; Functional Programming; Performance

Introduction

Big Data is centered on large amount of data what directly impacts the performances of the programs (e.g. to query a specific information) and then requires specific architectures to improve them [1], e.g. use of graph databases or distributed concurrent computations. Though a lot of technologies are available today to put Big Data into practice, theories usable to well understand the benefits/limitations of each architecture, to identify possible improvements or means to combine them are more rare [2]. In this context, the paper presents the capabilities offered by Category Theory together with a functional programming language (to implement the concepts and facilitate experimentation) to solve this limitation. In particular, it explains how functors can serve to model data structures (e.g. various representations of graphs) and how natural transformations can be used to change data structures or shift programs applicable to a particular data structure to another program for an other data structure. The concept of natural isomorphism then establishes to prove that two data structures represent the same information, or

that two programs are equivalent. Next, the equations representing the programs can serve to calculate computation steps (time complexity) and compare the performances of two equivalent programs, then show that a natural transformation is just an optimization. An advantage of Category Theory is to be easily and safely translated in most of the functional programming languages, what is interesting to make experiments and proposes new architectures or tools to Big Data community. As an illustration the paper proposes an optimized (by the way of natural transformations) implementation of an information server and its query language in the Haskell functional programming language. The other interests of the paper are then to detail the implementation and to give a comparison of the performances obtained with the standard tools available in the market. All the code is presented in the following parts what also confirms the fact that a functorial/functional approach leads to shorter programs than the ones developed in an other paradigms (imperative or object-oriented in particular). As a complement, it also shows than these "short" programs can implement complex algorithms (such as unification) by using the capabilities brought by the concepts (e.g. functors and higher-order functions). The comparison step finally shows that: 1) the program presented is able to deal with large sets of data, and 2) the program can be efficient - 5 to 100^{+} faster than other tools. The protocol used to get the measures (code and data transformation to other formats, for instance) is detailed.

The article is divided into 6 parts. The first part starts by introducing the contribution of the paper into context with "Related works". The second part entitled "Background" presents the fundamental concepts of Category Theory and how they can be translated into a functional programming language. In particular, this part details how "functors" can model data structures, and "natural transformations" changes of a program using a data structure to a new one, more efficient, using another structure. Then, it introduces models commonly used by information systems (relational, document or graph oriented). The "Method" part explains how Category Theory can be used to define an efficient information server and its query language (based on unification). The "Results" part presents the dataset considered in the experiments and gives a comparison of the performances (time to answer a query) obtained with the system proposed and standard tools (Sqlite, Mysql, Mongodb, Neo4j). The fifth part entitled "Discussion" examines in detail the benefit/limitation of the elements proposed, and shows how to apply them in other contexts. Finally, a conclusion summarizes the main elements presented: a theoretical approach of Big Data ; i.e. an efficient information system with a comparative study, and describes some of the perspectives considered.

Related works

Big Data is centered on very large datasets and a sample illustration is presented in the Figure 1. As explained in [3], dealing with a huge amount of data requires specific architectures both for hardware (e.g. cloud computers) and for software (e.g. graph database servers). Though many theoretical models are then proposed to get a plus value from all the data available, theories able to formalize the concepts under the tools commonly used to manage or query the data are more rare. The aim of this paper is to explain how Category Theory can solve this limitation and, associated

to a functional programming language, be used for instance to propose efficient information servers for large datasets (e.g. reducing the 0.67s in the Figure 1) or to shift data between various formats (this, by combining natural transformations).

Of course, the use of Category Theory for software development is not new. In particular, this theory has already shown its advantages in the domain of "program calculation" with for instance [4] or [5], in the domain of Model-Driven Engineering [6], etc. The concepts of the theory has also been implemented in some programming languages, such as ML, and can be used directly in these languages, e.g. [7]. At an extreme, the concepts have themselves been used to define a specific programming language in [8]. Category Theory has also lead to specific platforms for the management of (graph) data models [9] and query [10].

The contribution of this paper is to go further considering big datasets - what has not been considered in the above works, and by making possible the interchange of data with more classical tools found in the Big Data community.

Background

Elements of Category Theory

Category Theory is a field of mathematics introduced by McLane and Eilenberg to deal with "structures" (sets, graphs, algebras, etc.). This one defines general concepts such as categories (that can be viewed as labeled directed graphs describing a mathematical structure), functors (as relations between two categories) and also natural transformations/isomorphisms (as relations between two functors).

Categories and functors

A category is defined by a set of objects, a set of morphisms between this objects, a composition operator written ($\circ$) for morphisms and an *id*entity morphism for each object [11]. The composition is associative and has id as neutral element. A concrete example is given by the category $\mathcal{S}et = (X_i, f_j : X_k \to X_l, \circ, id_i)$ where objects correspond to sets $(X_i)_{i \in I}$, and morphisms to functions $(f_j)_{j \in J}$. This category can be easily related to (functional) programs by considering that sets model basic datatypes (e.g. boolean, integer, etc.), and morphisms (e.g. f_j) correspond to programs with a parameter X_k and a result X_l [12].

A functor F is a structure preserving map between two categories, i.e. it preserves composition $F(f_1 \circ f_2) = F(f_1) \circ F(f_2)$, and identities $F(id_{X_i}) = id_{F(X_i)}$. A well known example of a functor is the powerset $\mathcal{P} : \mathcal{S}et \to \mathcal{S}et$ with $\mathcal{P}(X_i)$ the set of subsets of X_i and $\mathcal{P}(f_i)\{x_1, ..., x_n\} = \{f_i(x_1), ..., f_i(x_n)\}$. By considering programs, this one can serve to model collections and simple transformations - $\mathcal{P}(f_i)$ being viewed as a loop applying f_i to the elements of a set. Another example is the product (bi)functor $X_i \times X_j$ with $(f_i \times f_j)(x_i, x_j) = (f_i(x_i), f_j(x_j))$. By considering programs, this one can serve to model records.

The preceding functors can be composed to model more complex data structures. For instance, directed labeled graphs can be represented by a set of edges and a functor $G(N) = \mathcal{P}(N \times N \times N)$ where N represents a set of nodes and $N \times N \times N$ edges of the form (source,label,destination). With a function $f : N \to N'$, we can define a graph morphism $m(f) = \mathcal{P}(f \times f \times f)$ that changes the nodes by preserving the structure of the graph - i.e. if (x, y, z) is an edge of g then $(f(x), f(y), f(z))$ is

an edge of $m(g)$. Now, it is easy to check that $m(id_N)$ is a identity morphism, morphisms are composable (i.e. $m(f \circ g) = m(f) \circ m(g)$) what makes the set of graphs and morphisms another example of a category called $\mathcal{G}raph$.

Transformations and optimization

A natural transformation corresponds to a relation between two functors. As an illustration, the graphs mentioned above can be represented in a different way by considering the functor $G'(N) = \mathcal{P}(N \times \mathcal{P}(N \times N))$ that associates adjacent links to each node. The relation between G and G' can then be represented by a natural transformation $\eta : G'(N) \to G(N)$. This one can be defined, by using set comprehension notation, as: $\eta(g') = \{(x, y, z) \,|\, (x, ys) \in g, (y, z) \in ys\}$. This transformation is invertible and the functors/datatypes are then said to be naturally isomorphic $G'(N) \cong_\eta G(N)$.

Now, if the two structures represent a "same" information, the performance of a program depends on the structure selected. As an example, a function/program to get the adjacent links, i.e. $get(n) : G(N) \to \mathcal{P}(N \times N)$, will have a complexity $\mathcal{O}(n)$ where n is the number of edges when using G, and $\mathcal{O}(m)$ where m is the number of nodes when using G', and $m \leq n$. So, $get'(n) : G'(N) \to \mathcal{P}(N \times N)$ is "faster" than $get(n)$. The change from G to G' can be viewed as an optimization technique called "memorization" in the sense that G' memorizes the result (i.e. adjacent links) for each input node and then eliminates extra computations [13]. The optimized version of the program will be obtained with $get'(n) = get(n) \circ \eta^{-1}$ that can be simplified by using the definitions of g and η^{-1} (and is known as short-cut fusion optimization [14]). Another common optimization technique consists in splitting data and use parallel computations. In the example of graphs and by considering a pair of computers, this can be modeled with $G''(N) = G(N) \times G(N)$. The function to get the adjacent links will be now $get''(n)(g_1, g_2) = \cup \circ (get(n)(g_1) \times get(n)(g_2))$ with a complexity $\mathcal{O}(max(n_1, n_2))$ where n_i is the size of g_i. And finally, we get an optimization chain that can be represented by: $\mathcal{O}(get'') \leq \mathcal{O}(get') \leq \mathcal{O}(get)$.

Implementation in Haskell

The elements presented can be easily translated in most of the functional programming languages, with in particular `Haskell` [15]. Products are then interpreted as pairs `(x,y)` and are associated to the higher-order function `mult f g (x,y) = (f x,g y)`. Powersets are replaced by lists `[x]` inductively defined by the empty list `[]` and a binary operator `(:)` to add an element to a list. $\mathcal{P}(f)$ is then represented by the `map f` function. Data structures and functors are then encoded as type synonyms, e.g. `[[x]]` to model a table or an array, `[(x,x,x)]` or `[(x,[(x,x)])]` to model a graph, etc. Natural transformations simply correspond to functions and the following code gives the example of an encoding in Haskell of η (eta) transformation detailed previously. The `concat` function concatenates a list of lists, and the dot `(.)` represents function composition. The function `get` returns the adjacent links, and `eta'` is the inverse of `eta` (the extra parameter `xs` represent the nodes' list in the graph `g`).

```
type G  x = [(x,x,x)]
type G' x = [(x,[(x,x)])]
```

```
eta :: G' x -> G x
eta g' = concat (map f g')
 where f (x,yz) = map (g x) yz)
       g x (y,z)= (x,y,z)

get :: x -> G x -> [(x,x)]
get n g = concat (map f g)
 where f (x,y,z) = if (x==n) then [(y,z)] else []

eta':: [x] -> G x -> G' x
eta' xs g = map f xs
 where f x = (x,get x g)

get':: x -> G' x -> [(x,x)]
get' n []         = []
get' n ((x,yz):xs) = if (x==n) then yz else get' n xs
```

Standard Data Models

There are 3 common data structures used to organize information in a Big Data context [16] with relational databases (e.g. Sqlite or Mysql), document-oriented databases (e.g. Mongo), and graph-oriented databases (e.g. Neo4j).

Relational models

Relational databases generally use a set of tables as illustrated in Figure 2, and relational algebra as a mathematical foundation. From a technical point of view, the standard language to manage a database or query a specific information is the Structured Query Language SQL [17]. The most fundamental constructs of the language are : 1) the creation of a new table, 2) the insertion of a new record in a table, and 3) the search/selection of an information in a table.

A table corresponds to a mathematical relation R, i.e. a subset of a cartesian product $R \subseteq X \times Y$ where X (resp. Y) corresponds to the first (resp. second) column. A query can then be represented by using set-comprehension notation, and a general example is $\{f(x,y) \,|\, (x,y) \in R, p(x,y)\}$ where p is a predicate ("where" clause in SQL) and f is a projection or transformation function [18]. This kind of queries has been studied in the literature with, for instance, [19] that uses functors, associated to two specific natural transformations (to define what is called a "monad"), to both gives a formal interpretation of queries and to study possible optimizations (by reducing computation steps). As a remark, one contribution of this paper is then in showing the impact of the data structures (functors) considered to interpret a query and in explaining how natural transformations can serve to new optimizations. Another contribution is in introducing pattern matching to simplify the definition of the queries.

Document-oriented models

Documents oriented databases commonly use tree structures as illustrated in Figure 3. An information is then obtained by its path from the root. The most important languages are here the eXtensible Markup Language (XML) and the Javascript Object Notation (JSON) used by Mongo databases [20].

Mathematical models are proposed in the literature with, for instance, [21] that use particular functors (called "monads") to represent the various constructs of a

query language over tree structures and their interpretation. We remind that a tree is an acyclic labeled directed graph, and the answer to a particular query is then a set of paths in this graph. The query language then defines simple queries (e.g. to test a node), combinators to compose other queries (e.g. sequential compositions or choices) or to repeat recursively a query other the children of a node. Here, a contribution of this paper is then to be centered on graphs (what naturally includes trees) and to use a declarative style, rather than an imperative one, for queries.

Graph-oriented models

Graphs oriented databases [22] use nodes and links between nodes as illustrated in the Figure 4. From a technical point of view, there exists a set of standards such as the Resource Document Framework (RDF) that is based on XML format or more specific languages such as the Cypher language used by the Neo4j tool [23]. Querying a particular information then consists in finding a morphism from the graph representing a query to a graph database [24]. Indeed, a query such as "(X is-city mulhouse) and (X has-latitude Y)" can be viewed as a labeled graph with two edges $\{city : X \rightarrow mulhouse, latitude : X \rightarrow Y\}$, where X/Y denote variables as illustrated in the right part of the Figure 4

From a more theoretical point of view, and by re-using the definitions of the functor representing graph, a query is a graph $G(N \cup X)$ where X is a set of variables. The result of a query is then a set of sets of pairs, e.g. $\{\{(X, 959679), (Y, 47.73)\}\}$, and the program finding the possible morphisms can be formalized by a function $unify : G(N') \times G(N) \rightarrow \mathcal{P}(\mathcal{P}(X \times N))$ where $N' = N \cup X$ is the union of constant nodes N plus variables X. As a remark, $\mathcal{P}(X \times N)$ is here a shortcut for a mapping function $f : X \rightarrow N$, and is generally called "environment" (this concept will be detailed in the next section). A contribution of the paper is then in the proposition of an efficient implementation of the $unify$ function extended to define a query language for the information server proposed.

More generally, graphs have many formalizations and applications in computer science. In particular, a detailed description of the above elements (graph matching and logic) from a categorical point of view can be found in [25]. Another application of Category Theory and graphs is given by graph rewriting systems [26] where the first step consists in finding a subgraph (left hand side of a rewriting rule) and a morphism. A contribution of the paper is, as explained above, in the proposition of an efficient unification algorithm to the preceding problem, and our actual works try to adapt the concept of "rewriting" to the one of "inference" (e.g. to create new information).

To conclude, the performances of these various data models and technologies can be found in the literature with in particular: [27] that describes the performances of the main tools using SQL and the ways to improve them, and [28] that studies the performances of graph query languages. There are also some comparison studies between these models with for instance [29] that compares the performances of SQL and Mongo databases, or [30] for a comparison between Neo4j and Mysql.

Method

Relations and comprehensions

As explained in the previous part, functors representing lists (collections) and products (records) can be composed to model data structures in various ways. In partic-

ular, a table used in a relational database can be abstracted by a two dimensional array, i.e. `[[x]]` in `Haskell` where x represents the type of the cells. A standard format to represent tabular data is the Comma Separated Values (CSV) format. Now, it is easy to define functions to read/write data in this format in Haskell as explained below. This code is detailed because its introduces specificity of the Haskell language used in the next sections, and is similar to other functions to read/write other formats (e.g. JSON) not presented for clarity reasons. As explained latter, these functions serve to the definitions of isomorphisms between functorial models implemented in Haskell and more standard models.

An interpreter for CSV can be defined in Haskell as follows and by considering the simplified grammar:

```
<val> := 'a' | ... | 'Z' | '0' | ... | '9'
<csv> := (<val>*(,<val>*)*'\n')*
```

From the grammar, the code for the parser, with the Parsec library [31], can be derived as below. The notation `[x..y]` represents the enumeration form `x` to `y`, `++` is the list concatenation operator and the `do` notation is used for sequential composition. The "oneOf" function then returns a parser that check if the first element of an input text belongs to the parameter of the function. The "many" function represents the repetition of an element and corresponds to the * operator in the BNF expression.

```
val :: Parser Char
val = oneOf (['a'..'z']++['0'..'9']++...)

csv = many $
 do many val
    many $ do oneOf [',']
              many val
    oneOf ['\n']
```

The program is then extended to extract some information. The `x<-` operator simply introduces a new variable `v` that stores an information, and `return` specifies the result of the function. The `(:)` is, as mentioned before, the Haskell operator to add an element to a list.

```
csv = many $
 do v  <- many val
    vs <- many $ do oneOf [',']
                    many val
    oneOf ['\n']
    return (v:vs)
```

Next, the two following functions use this parser to read a csv file with "fromCSV" and get a two dimensional array of strings (`[[String]]` in Haskell), and generate a csv file from an array with "toCSV". As a remark, strings are lists of characters and consequently all the functions on lists are available on them (map, concat, etc.). In the code, "intercalate v" concatenates a list of elements adding "v" between them.

```
fromCSV file = do
 str <- readFile file
 let Right dta = parse csv "" str
 return dta

toCSV file dta = do
```

```
 let lines = map (intercalate ",") dta
 let str   = intercalate "\n" lines
 writeFile file (str++"\n")
```

Having access to CSV data files, and by supposing that the graph of the left part of the Figure 4 is stored in "cities3.csv" (NB. the dataset is fully detailed in the "Results" section), the following program can now be used to obtain the performances of the get/get' functions. More precisely, the code is compiled by using the Glasgow Haskell Compiler (see `ghc` below), and execution time is measured with the shell command `time`. More precisely, the read/parse takes 2.182s, the result of get is obtained in 0.042s, the data transformation $\eta' : G \rightarrow G'$ takes 0.260s, and the result of get' is obtained in 0.009s - what is 4 times faster than get.

```
-- Compilation: ghc --make paper.hs
-- Performance: time ./paper
tst = do
 g <- fromCSV "cities3.csv"
 let r  = get  "959679" g
 let g' = eta' [show x | x<-[1..3173959]] g
 let r' = get' "959679" g'
 print r'
```

As a complement, the list comprehension notation `[f x | x<-xs, p x]` can be used to easily transform or query a specific information in a list (or a list of lists representing a CSV data as above). Comprehensions represent a syntactic sugar for `(map f).(filter p) xs` ; see [32] for more details and properties of this construct. As an application of the previous elements, and by considering that the data of the right array of the Figure 2 are stored now in "cities.csv", the code below gives examples of comprehension/queries with in particular the transformation (see r3) used to pass from a tabular information "cities.fr" (Figure 2) to a graph and a list of edges "cities3.fr" (Figure 4):

```
comprehension = do
 dta <- fromCSV "cities.csv"
 let hs = head dta
 -- extract french cities
 let r1=[[cit,acc,reg,pop,lat,long]|[cou,cit,acc,reg,pop,lat,long]<-(tail dta)
                                   , cou=="fr"]
 -- get cities/country at the same latitude
 let r2 = [[cou,cit] | [cou,cit,acc,reg,pop,lat,long]<-(tail dta)
                     , (lat-value)^2<epsilon]
 -- transformation array -> triples
 let r3 = concat [ [[cit,"country",cou],[cit,"latitude",lat],...]
                 | [cou,cit,acc,reg,pop,lat,long]<-(tail dta)] ]
 writeCSV r3 "cities3.csv"
 -- get latitude of mulhouse
 let r4 = [lat | [cou,cit,acc,reg,pop,lat,long]<-(tail dta), cit=="mulhouse"]
 print r4
```

Graphs and unification

Having modeled n-ary relations by the way of a functor $[[x]]$, and having defined natural transformations establishing the iso-morphism with the CSV standard - $CSV \cong_{fromCSV} [[x]]$, we can now use the same model to represent graphs ; this, by adding an extra condition: $\forall db \in [[x]], \forall e \in db, size(e) = 3$.

Thus, each element corresponds to a labeled edge $(source, label, destination)$ or $(subject, verb, complement)$ with a logical point of view of the graph/database. And if, list comprehension can be used for queries, it is more interesting to define a (human readable) query language able to find more complex information (e.g. join between other informations), and that can be used without knowing anything about Haskell (i.e. queries are passed as a parameter of the compiled program). A simple language is the First Order Language that defines predicates and quantifiers over variables. As mentioned previously, query expressions can be represented by graphs as illustrated by the graph pattern in the Figure 4 ; this one corresponds to the logical expression $\exists X.\exists Y.(X\ city\ mulhouse) \wedge (X\ latitude\ Y)$. The expression can be simplified to an Haskell list `[["?X","city","mulhouse"],["?X","latitude","?Y"]]`, and an interpreter for this query has to find the possible values in an array of "facts" (as the ones contained in "cities3.csv" mentioned above). The values are then obtained with an "unification" algorithm, and if we have previously presented such an algorithm with an application to an inference system for the web [33], this article proposes now an optimized version of this one (as an application of natural transformations) and a comparative study of the performances with more standard tools.

The first thing to consider when answering a query is the concept of "environment" that stores the value of the variables. For instance, a description such as "n°959679 is the city mulhouse and has latitude 47.73" (a part of the database) and a query such as "what is the latitude (let's say Y) of the n° (let's say X) corresponding to the city mulhouse" must lead to "X=n°959679 and Y=47.73". An environment can be modeled by a map from variables to values. This one can then be specified by a set of functions to create a new environment (newMap), to add an element (put), to test if a variable belongs to the domain of the map (has) and to extract the value of a variable (get). Many implementations are possible such as association lists (as follow) but others, maybe more efficient, implementations are possible such as binary trees for instance. The following code then proposes a functor for maps and associative lists.

```
type Map x y = [(x,y)]
has :: Eq x => Map x y -> x -> Bool
get :: Eq x => Map x y -> x -> y
put :: Map x y -> x -> y -> Map x y
newMap = []
```

The second element to consider is the mean to distinguish variables from values in a particular construct. This can be specified by a predicate $isvar : x \rightarrow Bool$ where x is the union set of variables and values. For instance, x can be replaced by strings and a predicate can test if the first character is a question mark representing a variable. Then, we can define a function testing if an x, that is either a variable or a value, is "equal" to another element/value x' - this in a particular environment. The result is yes/no and a new environment eventually extended with a new variable $x \mapsto x'$ (if not already defined in the environment). This function is given below:

```
equal :: Eq x => (x->Bool) -> x -> x -> Map x x -> (Bool,Map x x)
equal isvar x x' env =
 if (isvar x) then
   if (has env x) then ((get env x)==x',env)
```

```
    else (True,put env x x')
  else (x==x',env)
```

The preceding function can be extended to compare two lists (having the same size): the first one being composed with variables or values (and defining a "pattern" we are looking for), and the second with values - see "equaln" below. This function recursively tests if there is a matching between the first element of each list (x, x'), if true then it continues with the reminder of each lists (xs, xs').

```
equaln :: Eq x => (x->Bool) -> [x] -> [x] -> Map x x -> (Bool,Map x x)
equaln isvar [x]    [x']     env = equal isvar x x' env
equaln isvar (x:xs) (x':xs') env =
 case equal isvar x x' env of
  (True,env') -> equaln isvar xs xs' env'
  _           -> (False,env)
```

The function can be extended again to find in a list of constructs (the second element in the following function "unify1") the ones that match a pattern. This function simply applies the previous function to all the elements of the second list by keeping only the ones that match and their corresponding environment. Thus the result is a list of environment matching the pattern.

```
unify1 :: Eq x => (x->Bool) -> [x] -> [[x]] -> Map x x -> [Map x x]
unify1 isvar xs []       env = []
unify1 isvar xs (x':xs') env =
 case equaln isvar xs x' env of
  (True,env') -> env':(unify1 isvar xs xs' env)
  _           -> unify1 isvar xs xs' env
```

As a sample application, the `main` program below returns the "959679" that is the index corresponding to the city "mulhouse".

```
main = do
 dta <- fromCSV "cities3.csv"
 let pat = ["?X","city","mulhouse"]
 let result = unify1 isvar pat dta newMap -- change
 print result
```

Finally, the function can be generalized with a list of patterns - the first element in the following function "unify", to be found in a list of elements representing the database. In the code, the expression such as `\x->y` denotes an anonymous function $f(x) = y$. Next, the function uses the previous function on the first pattern x then continues with the next patterns xs and the environments env' that match x; this is performed with the "map" function. All the results/environments satisfying the patterns are then concatenated with the "concat" function.

```
unify :: Eq x => (x->Bool) -> [[x]] -> [[x]] -> Map x x -> [Map x x]
unify isvar [x]    xs' env = unify0 isvar x xs' env
unify isvar (x:xs) xs' env =
 concat (map (\env'->unify isvar xs xs' env') (unify1 isvar x xs' env))
```

As an application, the query used as an introductory example at the beginning of this part can be encoded by: `q=[[?X, city, mulhouse], [?X, latitude, ?Y]]`, and the dataset by: `db=[...[n°959679, latitude, 47.73],... [n°959679, city, mulhouse] ...]`. The result of `query isvar q db newMap` will then be `[[(?X,n°959679), (?Y,47.73)]]`. Now, to study the performances of the preceding programs, two queries are considered: one representing a simple query to get

a specific element ($q1$), and one representing a "join" and a complex query ($q2$). With the dataset and hardware architecture detailed in "Results" section, the time requires to return the result of $q1$ is 0.090s, and 0.170s for $q2$.

As a final remark, the dataset is represented here as a list of triples what is a common representation in Big Data community [34], but any tuple structures can be considered.

Optimization

Before illustrating on how natural transformations can be used for optimization, the previous implementation must be improved to eliminate the file loading/parsing from the measures (approx. 2.230s with the architecture detailed in the "Results" section), and thus consider only in-memory data. To proceed, the program has been transformed into a service (what is also the first step to parallel and distributed computations - see functor G'') with the code below. More precisely, `slave` will load a data `file` and listen to a given `port` of localhost. It continuously waits for a pattern (e.g. $q1$ or $q2$) to query, then calls the $unify$ function to finally send the result. The function/program `query` simply opens a connection to a host `h` at a port `p` and transmit a pattern `q`. The `main` program is just an utility function usable either to start a slave or to send a query. Its usage is explained bellow.

```
--- Usage: ./paper slave cities3.csv 9000 &
---        ./paper query localhost 9000 '[["?X","city","mulhouse"]]'
main = do
 a <- getArgs  -- returns command line arguments
 case a of
  ["slave", file, port]    -> slave file (read port :: PortNumber)
  ["query", host, port, q] -> query host (read port :: PortNumber) q

slave file port = withSocketsDo $ do
 db   <- fromCSV file
 sock <- listenOn $ PortNumber port
 slavebody sock db

slavebody sock db =
 forever $ do
  (handle, host, port) <- accept sock
  rc <- hGetLine handle
  let q = read rc :: [[String]]
  let r = unify isvar q db newMap
  hPutStrLn handle (show r)

query h p q = withSocketsDo $ do
 let n = connectTo h (PortNumber p)
 hd <- n
 hPutStrLn hd q
 rep <- hGetLine hd
 putStrLn rep
```

The preceding implementation is based on the functor G with a list of triples for the database. As mentioned in the presentation of Category Theory, natural transformations can be used to change both the functor and the performance of a program. In particular, it has been shown that the use of G' firstly (and G'' secondly) can dramatically reduce time complexity.

Thus, if the transformation $\eta : G \rightarrow G'$ have been explained by the way of a *get* function returning the adjacent links of an element, they can be generalized and applied to the $unify$ program. To proceed, the database "cities.csv" (based on the G model/functor) is transformed into a new base "ocities.hs" (using G' and obtained with η). Then, the functions "equals" and "unify" are changed to use a dataset of type G' and thus the function get' at the place of *get* - the new functions are prefixed with "o" for "optimized". In particular, the form of the pattern is now taken into account and a pattern beginning with a constant element e no need to call the unify function in the sense the possible values to be considered by the rest of the pattern are directly given by $get'\ e\ g'$. With the optimized version of the functions, the time required to get the result of $q1$ is now 0.020s (4 times faster than the original version and what is the same factor than the one obtained for *get*), and 0.040s for $q2$ (4 times faster than the unoptimized version).

```
oequaln isvar [x,y,z] g' env =
 if not (isvar x) then
   let dbs = get' x g'
       res = unify1 isvar [y,z] dbs env
    in res
  else if not (isvar y) then
         let dbs = get' y g'
             res  = unify1 isvar [x,z] dbs env
          in res
        else []

ounify isvar [x]    g' env = oequaln isvar x g' env
ounify isvar (x:xs) g' env =
 concat (map (\env'->ounify isvar xs g' env') (oequaln isvar x g' env))
```

Next, the functor $G'' = G \times G$ (resp. with the optimized version of the unify algorithm we can consider also $G^{(3)} = G' \times G'$) is considered to both split a dataset and use concurrent executions. By re-using the "slave" and "query" programs, it is easy now to check the performances without forgetting that, if the model is theoretically more efficient, it involves in practice new elements such as communication times. More precisely, the distributed version of the application is defined as follow (each part contains half of "cities3.csv").

```
./paper slave cities3-part1.csv 9000 &
./paper slave cities3-part2.csv 9001 &
time (./paper query localhost 9000 q &
      ./paper query localhost 9001 q)
```

The time required now is then 0.055s for $q1$ with G'' and "unify" (what represents a gain of 0.035s) and 0.015s with $G^{(3)}$ and "ounify" (gain of 0.005s). For $q2$ the respective time are 0.100s for $q2$ in the unoptimized version of "unify" (gain of 0.070s), then 0.025s in the optimized version (gain of 0.015s).

Concrete syntaxes

At this stage, an optimized information server with a unification based query language is proposed. To compare the performance of this one to the standard tools commonly found on the market, natural isomorphisms have to be defined with: $G \approx_{toSql} SQL$, $G \approx_{toMongo} JSON$ and $G \approx_{toNeo4j} Cypher$ - what corresponds respectively to relational, document and graph oriented models. As a remark, by having previously formalized the isomorphism $G \approx_{toCSV} CSV$, the code

presented can serve to easily define transformations between various formats, e.g. $toSQL \circ fromCSV : CSV \rightarrow SQL$ (see `mkSql` below).

Thus, the dataset has been translated into SQL statements, by using the following program/transformation and the result stored in the "cities3.sql" file. This latter has been used to fill a Sqlite database with: `cat cities3.sql | sqlite3 database.db`. Finally, the performance for an equivalent expression of $q1$ has been obtained with: `time (echo "SELECT src FROM Db WHERE dst LIKE 'mulhouse' AND lbl LIKE 'city';" | sqlite3 database.db)` - the value is 0.040s. For $q2$, the measure is approximately 0.680s and is obtained with: `SELECT D.dst FROM Db as D LEFT OUTER JOIN Db as D2 ON D.src = D2.src WHERE D2.lbl LIKE 'city' AND D2.dst LIKE 'mulhouse' AND D.lbl LIKE 'latitude';`.

```
toSql db = concat [ "CREATE TABLE Db (src text,lbl text, dst text);\n", db']
 where
  db' = concat (map (\e->concat ["INSERT INTO Db VALUES (",fmt e,");\n"]) db)
  fmt []     = ""
  fmt [x]    = concat ["'",x,"'"]
  fmt (x:xs) = concat ["'",x,"',",fmt xs]

mkSql = do
 dta <- fromCSV "cities3.csv"
 writeFile "cities.sql" (toSql dta)
```

The SQL configuration file has also been used with a Mysql server and the performances obtained are here 0.240s for $q1$ and 0.500s for $q2$.

The preceding approach has been used again for Mongo with another translation program. More precisely, the dataset has been translated in Javascript statements stored in "cities3.js". This file is loaded by using the "load('cities3.js')" inside the Mongo console. Then the equivalent query for $q1$ is then obtained with: `mongo -eval "db.store.find(lbl:'city', dst:'mulhouse').shellPrint()"`, and is executed in 0.160s. Mongo being not well suited to expression join, $q2$ has not been considered.

```
toMongo db = db'
 where
  db' = concat (map (\e->concat ["db.store.insert(",format e,")\n"]) db)
  format [x,y,z]     = concat ["{ src:'",x,"', lbl:'",y,"', dst:'",z,"' }"]
```

As a final comparison, the dataset has been tested with the Neo4j graph database and its query language Cypher. First, the dataset has been loaded from the CSV file with the following command from the Neo4j interface.

```
LOAD CSV WITH HEADERS FROM "file:///cities3.csv" AS line
MERGE (n:Node {name: line.src})
CREATE (m:Node {name: line.dst})
CREATE (n)-[:Link {name: line.lbl}]->(m)}
```

Next, the query $q1$ is obtained in 0.850s with: `echo "MATCH ((x)-[r:Link {name:'city'}]->(y)) WHERE y.name='mulhouse' RETURN x.name;" | cypher-shell -u user -p password`.

Query $q2$ is expressed by: `MATCH (x {name:'mulhouse'}),(y),(z) WHERE (y)-[:Link {name:'city'}]->(x) AND (y)-[:Link {name:'latitude'}]->(z) RETURN z.name;` and is obtained in more than one hour !

Results

Dataset, architecture and queries considered

The dataset [35] considered to study the performance of the various systems, and compare their performances, is the one presented in Figure 4 and `cities.csv` in the code. It consists in $3.10^6 \times$ `[Country, City, AccentCity, Region, Population, Latitude, Longitude]` representing data for the most important cities around the world ; what also corresponds to a file size of 150Mo. The computer(s) used for performances' measurement is an EliteBook(s) 820 with processor(s) Intel i5 with 4 threads at 2.3GHz, and 16Go RAM. The two queries taken into account for comparing performances are: 1) a simple query $q1 = [[?X, city, mulhouse]]$ and 2) a complex/join query $q2 = [[?X, city, mulhouse], [?X, latitude, ?Y]]$.

Performances

The various performances for each system (sorted by $q1$) is presented in the Figure 5. As mentioned in the introduction, the answer to a "basic" query ($q1$) is obtained in 20ms, what is 2 times faster than a Sqlite server, more than 8-10 times faster than Mysql and Mongo, and more than 40 times faster than Neo4j. For a more complex query involving a join ($q2$), the tool proposed is approximately 10^+ times faster than Sqlite/Mysql, and is really more faster than Neo4j (more than 1 hour to get a result).

As a concluding remark, the use of an unification algorithm leads to simpler queries as shown in the Figure 6.

Discussion

If the elements proposed (functorial modeling and sample information server) could be seemed as being focused on a specific dataset and application, they can be easily generalized or applied to other contexts.

For instance, the search of documents containing a keyword (e.g. Google search) can be simply modeled as follow: the database consists in a set of documents D having each one a set of words W what corresponds to the functor $[D \times [W]]$; this one can be transformed to $[W \times [D]]$ as illustrated in Figure 7 and what corresponds to *indexation*. Then, the *get* function, that simply return the value associated to a key in an association list (see the "Map" functor), can serve to find a specific word to return the set of documents containing this word. The various functions in the Figure are natural transformations that are composed to define the *search* function. As shown in the Figure, the performance of the search using the initial data structure $[D \times [W]]$ is intuitively worst than the one using $[W \times [D]]$ and *get*.

Such a model can be, as explained in this paper, implemented directly in most of the functional programming languages to get a concrete application and then compare the performances of this latter to the standards. For instance, one of our study has considered 119767 page links on a set of documents from Wikipedia [36] Two queries have then been considered with a simple one $q1$ to find the pages having a particular target page (e.g. $(Zulu, ?X)$), and a more complex one $q2$ to get the intermediate pages between two pages (e.g. $(Zulu, ?X) \wedge (?X, US)$). The time to get the answer is then presented in Figure 8 where the database db (resp. db', etc.) is encoded with the functor G (resp. G', etc.) presented in the paper.

A comparison with the performances of more standard tools is proposed on Figure 9. These values are obtained by using the natural isomorphisms presented in the paper (e.g. $G \approx_{toSQL} SQL$) and stay globally equivalent to the ones already presented in Figure 5 (with another dataset).

Conclusion

This article describes on how Category Theory combined with a functional programming language can be interesting in a Big Data context. More precisely, it has explained how the concepts of functors (G^i) and natural transformations ($\eta^i : G^j \rightarrow G^k$) can serve to represent data structures and data transformations usable to optimize programs (and make them more efficient particularly when they have to deal with a large set of data). It also explains how to implement the concepts and then propose as a result an efficient information server using a logical query language based on "unification" (and graph morphisms) to facilitate the search of an information. Finally, the paper analyzes the performance of this tool (time required to get the result of a query) and gives a comparison with standard databases: MySql (and Sqlite), Mongo and Neo4j. The natural transformations presented and applied to the program make it approximately 10^+ times faster (100^+ times faster for Neo4j) as shown in Figure 5.

Three complementary perspectives are considered. The first one consists in studying other data structures and functors commonly found in program development such as binary trees, for instance, and that have already shown their benefits in information search (e.g. complexity $\mathcal{O}(log(n)$ for search). The second perspective is an improvement of the programs presented in the paper that corresponds to a prototype and not an "industrial" tool. To proceed, three elements will be considered with: 1) errors management and all the code required to get a robust tool, 2) a command language to interact with the server and facilitate the database management (e.g. load/save), and 3) an access to the server through the web, i.e. anybody will be able to use it. The final perspective will be in the use of the optimizations proposed by the tools used for comparisons (e.g. indexation in Mysql) to get a better comparison of the performances. This perspective is complementary from the first one and must help to formalize, by the way of functors and natural transformations, the general principles already used by today technologies (and maybe proposes other improvements).

Ethics approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Availability of data and material
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Funding
Not applicable.

Author's contributions
All mentioned authors contribute in the elaboration of the article. All authors read and approved the final manuscript.

Author's information
Laurent Thiry is Professor of Computer Science at University of Mulhouse (France). His main research interests are Software and Model-Driven Engineering, Formal Methods and Functional Programming for complex software, He has published several research articles in peer-reviewed international journals and conferences, and has served several conferences as a program chair, on these topics. The elements proposed result mainly from its participation to various international, european or national projects. Dr Laurent Thiry is the corresponding author and can be contacted at: laurent.thiry@uha.fr.
Heng Zhao is a second year PhD Student working on the application of Category Theory's concepts to the software engineering (data modeling and transformations for efficient applications). Heng Zhao can be contacted at: heng.zhao@uha.fr.
Michel Hassenforder is Full Professor of Computer Science at University of Mulhouse (France). His research interests are Software Engineering, Information Systems and Programming Languages. He has published several research articles in peer-reviewed international journals and conferences, and has participated to many international, european or national projects, on these topics. Michel Hassenforder can be contacted at: michel.hassenforder@uha.fr.

Acknowledgments
Not applicable.

References
1. Chen, M., Mao, S., Liu, Y.: Big data: A survey. Mobile Networks Application **19**(2), 171–209 (2014)
2. Erl, T., Khattak, W., Buhler, P.: "Big Data Fundamentals: Concepts, Drivers and Techniques. Prentice Hall Press, Upper Saddle River, NJ, USA (2016)
3. Kune, R., Konugurthi, P.K., Agarwal, A., Chillarige, R.R., Buyya, R.: The anatomy of big data computing. Software, Practice and Experience **46**(1), 79–105 (2016)
4. Fokkinga, M.M.: Calculate categorically! Formal Aspects of Computing **4**(1), 673–692 (1992)
5. Meijer, E., Hutton, G.: Bananas in space: Extending fold and unfold to exponential types. In: Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture. FPCA '95, pp. 324–333. ACM, New York, NY, USA (1995)
6. Thiry, L., Hassenforder, M.: A calculus for (meta)models and transformations. International Journal of Software Engineering and Knowledge Engineering **24**(5), 715–730 (2014)
7. Rydeheard, D.E., Burstall, R.M.: Computational Category Theory. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK (1988)
8. Hagino, T.: A categorical programming language. PhD thesis (1987)
9. Spivak, D.I.: Ologs: a categorical framework for knowledge representation. CoRR **abs/1102.1889** (2011)
10. Wisnesky, R.: Functional query languages with categorical types. PhD thesis, Cambridge, MA, USA (2014)
11. Goguen, J.A.: A categorical manifesto. In: Mathematical Structures in Computer Science, pp. 49–67 (1991)
12. Barr, M., Wells, C. (eds.): "Category Theory for Computing Science, 2nd Ed.". Prentice Hall International (UK) Ltd., Hertfordshire, UK (1995)
13. Okasaki, C.: "Purely Functional Data Structures". Cambridge University Press, New York, NY, USA (1998)
14. Bird, R., de Moor, O.: Algebra of Programming. Prentice-Hall international series in computer science. Prentice Hall, Upper Saddle River, NJ, USA (1997)
15. Shukla, N.: Haskell Data Analysis Cookbook. Packt Publ., Birmingham (2014)
16. Furht, B., Villanustre, F.: "Big Data Technologies and Applications", 1st edn. "Springer Publishing Company, Incorporated", Switzerland (2016)
17. Groff, J., Weinberg, P.: SQL The Complete Reference, 3rd Edition, 3rd edn. McGraw-Hill, Inc., New York, NY, USA (2010)
18. Trinder, P.: Comprehensions, a query notation for dbpls. In: Proceedings of the Third International Workshop on Database Programming Languages : Bulk Types &Amp; Persistent Data: Bulk Types &Amp; Persistent Data. DBPL3, pp. 55–68. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1992)
19. Wadler, P.: Comprehending monads. In: Proceedings of the 1990 ACM Conference on LISP and Functional Programming. LFP '90, pp. 61–78. ACM, New York, NY, USA (1990)
20. Chodorow, K., Dirolf, M.: MongoDB: The Definitive Guide, 1st edn. O'Reilly Media, Inc., Switzerland (2010)
21. Gottlob, G., Koch, C.: Monadic queries over tree-structured data. In: Proceedings 17th Annual IEEE Symposium on Logic in Computer Science, pp. 189–202 (2002)
22. Robinson, I., Webber, J., Eifrem, E.: Graph Databases. O'Reilly Media Inc., Switzerland (2013)
23. Vukotic, A., Watt, N., Abedrabbo, T., Fox, D., Partner, J.: Neo4J in Action, 1st edn. Manning Publications Co., Greenwich, CT, USA (2014)
24. Wood, P.T.: Query languages for graph databases. SIGMOD Rec. **41**(1), 50–60 (2012)
25. Courcelle, P.B., Engelfriet, D.J.: Graph Structure and Monadic Second-Order Logic: A Language-Theoretic Approach, 1st edn. Cambridge University Press, New York, NY, USA (2012)
26. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations. World Scientific Publishing Co., Inc., River Edge, NJ, USA (1997)
27. Gulutzan, P., Pelzer, T.: SQL Performance Turning. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
28. Holzschuher, F., Peinl, R.: Performance of graph query languages: Comparison of cypher, gremlin and native access in neo4j. In: Proceedings of the Joint EDBT/ICDT 2013 Workshops. EDBT '13, pp. 195–204. ACM, New York, NY, USA (2013)
29. Parker, Z., Poe, S., Vrbsky, S.V.: Comparing nosql mongodb to an sql db. In: Proceedings of the 51st ACM Southeast Conference. ACMSE '13, pp. 5–156. ACM, New York, NY, USA (2013)
30. Vicknair, C., Macias, M., Zhao, Z., Nan, X., Chen, Y., Wilkins, D.: A comparison of a graph database and a relational database: A data provenance perspective. In: Proceedings of the 48th Annual Southeast Regional Conference. ACM SE '10, pp. 42–1426. ACM, New York, NY, USA (2010)

31. Hutton, G., Meijer, E.: Monadic parsing in haskell. J. Funct. Program. **8**(4), 437–444 (1998)
32. Buneman, P., Libkin, L., Suciu, D., Tannen, V., Wong, L.: Comprehension syntax. SIGMOD Rec. **23**(1), 87–96 (1994)
33. Thiry, L., Mahfoudh, M., Hassenforder, M.: "a functional inference system for the web". "International Journal of Web Applications" **6**(1), 1–13 (2014)
34. Cur, O., Blin, G.: "RDF Database Systems: Triples Storage and SPARQL Query Processing, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2014)
35. http://download.maxmind.com/download/worldcities
36. http://snap.stanford.edu/data/wikispeedia.html

Figures

Figure 1 Sample performance.

Figure 2 Sample relational DB.

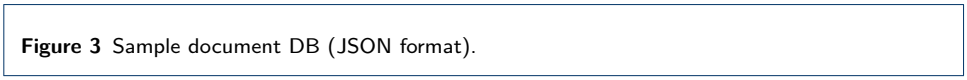

Figure 3 Sample document DB (JSON format).

Figure 4 Sample graphs and query/morphism.

Figure 5 Performances comparison.

Figure 6 Equivalent expressions.

Figure 7 Document search model.

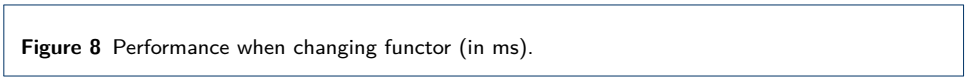

Figure 8 Performance when changing functor (in ms).

Figure 9 Performances comparison (cont.).